Reinforcing the Object-Oriented Aspect of Probabilistic Relational Models

Lionel Torti, Pierre-Henri Wuillemin, Christophe Gonzales

Reinforcing the Object-Oriented Aspect of Probabilistic Relational Models

Lionel Torti - Pierre-Henri Wuillemin - Christophe Gonzales
LIP6 - UPMC - France
firstname.lastname@lip6.fr

Abstract

Representing uncertainty in knowledge is a common issue in Artificial Intelligence. Bayesian Networks have been one of the main models used in this field of research. The simplicity of their specification is one of the reason for their success, both in industrial and in theoretical domains. The widespread use of Bayesian Networks brings new challenges in the design and use of large-scale systems, where this very simplicity causes a lack of expressiveness and scalability. To fill this gap, an increasing number of languages emerged as extensions of Bayesian Networks with many approaches: first-order logic, object-oriented, entity-relation, and so on. In this paper we focus on Probabilistic Relational Models, an object-oriented extension. However, Probabilistic Relational Models do not fully exploit the object-oriented paradigm, in particular they lack class inheritance. Using Object-Oriented Bayesian Networks as a basis, we propose to lightly extend PRMs framework resulting in stronger object-oriented aspects in probabilistic models.

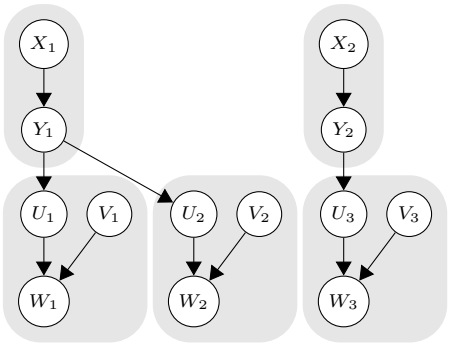
Probabilistic graphical models (Koller and Friedman, 2009) are a general purpose framework for dealing with uncertainty. Their applications to many different domains has stimulated an uninterrupted process of creation of new frameworks based on probability theory. Bayesian Networks (Pearl, 1988) are among the most popular framework for uncertainty in AI.

In recent years, the Statistical Learning community has actively proposed new probabilistic frameworks, closing the gap between first-order logic and probability theory (Getoor and Taskar, 2007). New models such as Object-Oriented Bayesian Networks (Koller and Pfeffer, 1997; Bangsø and Wuillemin, 2000a), Multiply-Sectioned Bayesian Networks (Yang, 2002), Probabilistic Relational Models (Getoor et al., 2007) and Multi-Entity Bayesian Networks (Laskey, 2008) have extended Bayesian Networks and widen their range of application.

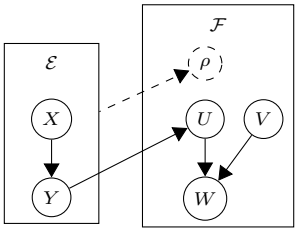
In many situations, these new first-order logic-based networks can be efficiently learned from databases and used for answering probabilistic queries. However, there are situations like nuclear plant safety problems where the scarcity of data available prevents such learning. For such problems, oriented graphical models such as Probabilistic Relational Models (PRMs) are often more suitable than the aforementioned first-order models because they can often be modeled by interactions with experts of the domain.

PRMs have an object-oriented basis, but they lack fundamental mechanisms related to class inheritance. In software engineering, such object-oriented designs has proved very useful for creating complex software. In this paper, we illustrate why these mechanisms are necessary for practical design of large-scale systems and we show how light extensions can enforce strong object-oriented features into the PRMs framework. In addition, we propose a representation of PRMs with such mechanisms using parfactors, the state-of-the-art framework for first-order probabilistic inference (Poole, 2003). All the concepts we present here are implemented in our open source C++ framework called *aGrUM* and can be represented in the SKOOL language (http://agrum.lip6.fr).

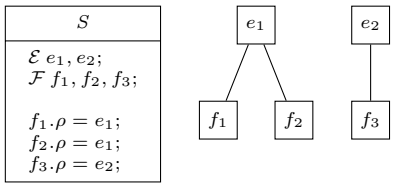
Throughout this paper, we will use an analogy with oriented-object programming in order to ease the presentation of our framework. It is organized as follows: after briefly introducing the classical PRM framework, we define the notions of attribute typing and type inheritance. Then we extend the notion of class inheritance with interfaces, to conclude



(a) A Bayesian network. The gray areas do not belong to the BN specification



(b) Two connected classes $\mathcal{E}$ and $\mathcal{F}$.



(c) The system declaration and the instance diagram corresponding to the BN of figure 1(a).

Figure 1: Representation of a BN as a PRM: analysis of the BN (a) reveals the use of two recurrent patterns, which are confined in two classes (b). Hence, a system equivalent to the BN may be built (c).

our contribution with the mechanisms for attribute and reference overloading. Finally we describe how PRMs with strong object-orientedness can be described using parfactors.

1 Description of PRMs

Fig. 1(a) shows a Bayesian Network (BN) encoding relations between two different kinds of patterns (variables X_i, Y_i on one hand and U_j, V_j, W_j on the other hand). We assume that the conditional probability tables (CPT) associated with variables with the same capital names are identical. When using PRMs, the main idea is to abstract each pattern as a generic entity, called a *class*, which encapsulates all the relations between the variables of the pattern. So, in Fig.1(b), $\mathcal{E}$ encapsulates precisely variables X_i and Y_i as well as their probabilistic relations (arc (X_i, Y_i)) and conditional probability distributions. The pattern of variables U_j, V_j, W_j cannot be directly encapsulated in a class since the CPTs of variables U_j are conditional to some variables Y_k (e.g., the CPT of U_3 is $P(U_3|Y_2)$ according to Fig.1(a)). Hence classes must have a mechanism allowing to refer to variables outside the class. In PRMs, this mechanism is called a *reference slot*. Basically, the idea is to create some function ρ connecting two classes and allowing both classes to access the variables of the other class. Now, as shown in Fig.1(c), the original BN can be built up from the PRM: it is sufficient to create two instances, say e_1 and e_2, of class $\mathcal{E}$ as well as three instances f_1, f_2, f_3 of $\mathcal{F}$ and connect them using one edge per reference slot. Note that there is no limit to the number of times an instance can be referenced (see e_1 in Fig.1(c)).

1.1 PRM-related definitions

In this section, we present the minimal set of definitions needed for the rest of the paper. The reader may refer to (Pfeffer, 2000) and (Getoor et al., 2007) for a more detailed presentation.

Definition 1 (Class)**.** A *class* $\mathcal{C}$ is defined by a Directed Acyclic Graph (DAG) over a set of attributes, i.e. random variables, $\mathbf{A}(\mathcal{C})$, a set of references (slots) $\mathbf{R}(\mathcal{C})$, and a probability distribution over $\mathbf{A}(\mathcal{C})$. To refer to a given random variable X (resp. reference ρ) of class $\mathcal{C}$, we use the standard Object Oriented notation $\mathcal{C}.X$ (resp. $\mathcal{C}.\rho$).

Definition 2 (Instance)**.** An *instance* c is the use (the instantiation) of a given class $\mathcal{C}$ in a BN. There are usually numerous instances of a given class $\mathcal{C}$ in a BN. Notation $c.X$ (resp. $c.\rho$) refers to the instantiation of $\mathcal{C}.X \in \mathbf{A}(\mathcal{C})$ (resp. $\mathcal{C}.\rho \in \mathbf{R}(\mathcal{C})$) in c. By abuse of notation, we denote the sets of such instantiations as $\mathbf{A}(c)$ and $\mathbf{R}(c)$ respectively.

Fig. 1(b) shows two classes, $\mathcal{E}$ and $\mathcal{F}$, with attributes $\mathcal{A}(\mathcal{E}) = \{X, Y\}$ and $\mathcal{A}(\mathcal{F}) = \{U, V, W\}$. There is also one reference in class $\mathcal{F}$ denoted by ρ which is used to define the dependencies between $\mathcal{E}.Y$ and $\mathcal{F}.U$. Such dependency is defined using a path, called a reference chain, from one attribute to another. In Fig. 1(b), the path representing the dependency between $\mathcal{E}.Y$ and $\mathcal{F}.U$ is $\mathcal{F}.\rho.Y$. More

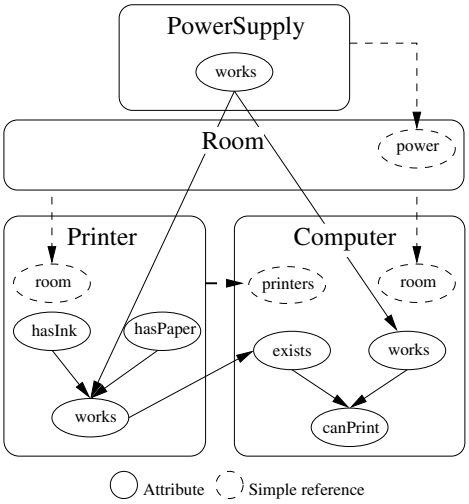


Figure 2: The printer example.

simply, $\rho.Y$ is said to be a parent of U in class $\mathcal{F}$.

Definition 3 (System, grounded net)**.** A system $\mathcal{S}$ is the representation of a BN as a set of class instances in which each reference has been linked to another instance. Conversely, the grounded network of a system $\mathcal{S}$ is the BN represented by $\mathcal{S}$.

As a consequence, in a system, each random variable $c.X$ is a copy of a random variable $\mathcal{C}.X \in \mathbf{A}(\mathcal{C})$ and is assigned a copy of the CPT assigned to $\mathcal{C}.X$ in $\mathcal{C}$. The difference between a system and a grounded net is that all structural information (classes, instances, references, ...) are lost when reasoning with a grounded net. Finally, PRMs are considered as an object-oriented formalism due to the encapsulation of attributes inside their classes. This feature is inherited from Object-Oriented Bayes Nets. Exploiting this encapsulation is the core of structured inference (Pfeffer, 2000).

1.2 Real Object-Oriented PRMs

The following discussion provides insight about how PRMs lack fundamental concepts of the object-oriented paradigm and how such concepts can greatly improve the representative power of PRMs. We will illustrate our point with a simple example of a printer breakdown diagnosis illustrated in Fig. 2. We consider a network with a power supply, black & white printers, color printers and computers. Printer's types, brands and ages vary from one printer to another. Printers and computers are placed in rooms and each computer is connected to every printer in the same room. All printers and computers are connected to the same power supply. Our main objective is to answer the following query: using a given computer, can I print? We can also think of other queries, not asked by a user but rather by an intervening technician: is a paper jam responsible for the printer's breakdown? Is the magenta cartridge of a color printer empty? Etc.

From the computer point of view, our system needs to take into account: (i) the fact that a printer prints in color is irrelevant for black & white printings; (ii) breakdowns can have different causes, but we only need to know whether printing is possible or not. From the technician perspective, we shall consider that: (i) different printers have different types of breakdowns, which can sometimes be partial, e.g. a color printer with no more cyan ink can still print in black & white; (ii) different types or brands imply different probabilities of breakdowns; (iii) the printer's features shall be taken into account since specific queries can be asked for each printer, e.g., can I print in color? Is the A3-tray empty? Etc.

These points of views force our system to be both generic (the computer's perspective) and specific (the technician's perspective). This is precisely why a strong object-oriented framework is needed. Let us do an analogy with computer programming. A class defines general concepts common to a family of objects. It is possible to define new concepts using inheritance: if class B inherits from class A, it inherits A's properties but can also specialize the concepts represented by A. Either by overloading A's attributes and methods (behavior specialization) or by adding new attributes and methods (functionality specialization). The next section proposes an extension of PRMs which will serve as a basis to strengthen class inheritance and we will show that this can be done with small and intuitive changes.

2 Attribute typing and type inheritance

Attribute typing arises naturally when using PRMs as a modeling framework: similarly to classes that represent repeated patterns in a system, an attribute type describes a family of random variables sharing the same domain. For instance, types *Boolean* and *state* would be the types of all the random variables

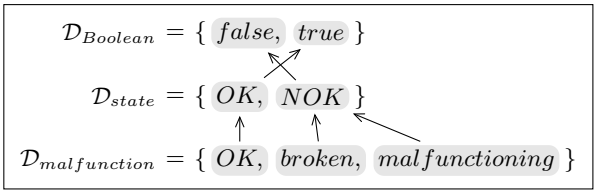


Figure 3: An illustration of type inheritance with attribute types *Boolean*, *state* and *malfunction*.

with domains $\{false, true\}$ and $\{OK, NOK\}$, respectively.

Definition 4 (Attribute typing)**.** An attribute type τ describes a family of distinct discrete random variables sharing the same domain $\mathcal{D}_\tau = \{l_1, \ldots, l_n\}$, where n is the domain size of τ.

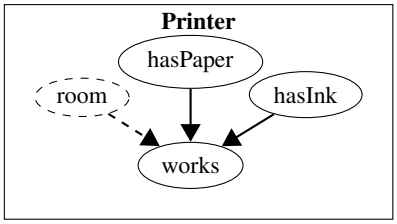
Types such as *Boolean* and *state* are frequently encountered when dealing with experts. For instance, they can be used to describe the states of equipments subject to breakdowns. In this case, type *state* enables a finer description of the possible failures than just the OK/NOK state. This can prove critical for some industrial applications: consider an air conditioner in a computer server room working improperly; then, assigning it state *malfunction* may help diagnose the servers malfunctions. Type *state* can be viewed as a *specialization* of type *Boolean*. Specializing general concepts into more specific ones is the goal of *inheritance*. Type inheritance is the process of decomposing labels into a partition of more specific and precise descriptions of a domain. To properly define this concept, we will need an additional notion, that of Domain Generalization Function (DGF):

Definition 5 (Domain Generalization Function)**.** A Domain Generalization Function (DGF) is a surjective function $\Phi : \mathcal{D}_\tau \rightarrow \mathcal{D}_\lambda$ where τ and λ are two distinct attribute types.

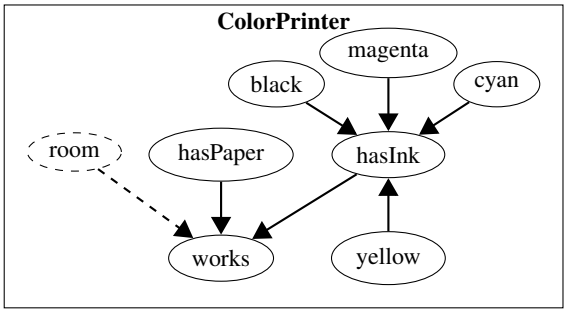
Obviously, given two distinct attribute types τ and λ, there exists a DGF $\Phi : \mathcal{D}_\tau \rightarrow \mathcal{D}_\lambda$ if and only if $|\mathcal{D}_\tau| \geq |\mathcal{D}_\lambda|$. DGFs will be used to define type inheritance in PRMs:

Definition 6 (Type inheritance)**.** An attribute type τ inherits from another attribute type λ if it is defined using a DGF $\Phi : \mathcal{D}_\tau \rightarrow \mathcal{D}_\lambda$.

Fig.3 illustrates type inheritance: in this figure, arcs represent the specialization of concepts. For



(a) Dependencies of the *Printer* class.



(b) Dependencies of the *ColorPrinter* class, which is a subclass of *Printer*.

Figure 4: Example of class inheritance. Dashed arcs represent dependencies with attributes in another class.

instance, attribute type *malfunction* has two labels, *broken* and *malfunction*, which are specializations of label *NOK* of attribute type *state*. As is, attribute inheritance is only a semantic relation: *state*'s label *OK* is a sort of *true*, *broken* is a sort of *false*, etc. We will show how to exploit such concepts probabilistically in the following sections.

3 Classes and interfaces

As in oriented-object programing, class inheritance in PRMs starts by a copy of the super class into its subclass. This implies that all attributes, references, dependencies, i.e. arcs, and CPTs are copied into the subclass. However, the content of the super class is only a basis for the subclass, as new attributes, references and dependencies can be added to the inherited structure. The first definitions of class inheritance for probabilistic models can be found in (Koller and Pfeffer, 1997) and (Bangsø and Wuillemin, 2000b). Note that these definitions differ greatly. In this paper, we propose some extensions of the work by Bangsø and Wuillemin.

3.1 Class inheritance

Fig. 4 illustrates class inheritance on the printer example of Fig. 2. Here, we introduced a new class, namely *ColorPrinter*, which is a subclass of *Printer*. Fig. 4(b) is a representation of the *ColorPrinter* class dependencies. This example suggests several remarks: (i) all the attributes and references belonging to class *Printer* also belong to *ColorPrinter*; (ii) new attributes have been added; (iii) attribute *ColorPrinter.hasInk* has additional parents (and thus a new CPT).

The first remark is similar with oriented-object programming languages: a subclass inherits the attributes and references of its super class. This implies that when an element is not overloaded, it is not necessary to redeclare it. The second remark is the functionality specialization of class inheritance: by adding new attributes, a subclass becomes more specific and offers new possibilities for entering evidence and submitting queries. In Fig. 4(b), attributes *black*, *magenta*, *cyan* and *yellow* represent the different kinds of inks used in a color printer, a feature that is not necessarily present in all printers. The third and fourth remarks are examples of attribute overloading, which consist of: (i) enabling changes in the values of the attribute's CPTs; (ii) adding or removing parents; (iii) overloading an attribute's type (this point is explained below).

3.2 Interface implementation

In modern programming languages, interfaces are used to handle multiple inheritance and to manipulate objects at a high abstract level. They define a set of methods which are guaranteed to exist in any class implementing them. Note that interfaces do not provide the bodies (the execution codes) of these methods but only their signatures. An interface in a PRM follows the same principle: it is a set of attributes and references; it defines neither probabilistic dependencies nor CPTs. As in programming languages, a PRM interface cannot be instantiated.

A PRM interface can be used to define dependencies between classes using abstraction: given two classes X and Y, if Y has an attribute depending on an attribute of X, then the only information needed is the type of X's attribute. As a consequence, the minimal set of information required to define probabilistic dependencies is composed of references and attribute's types.

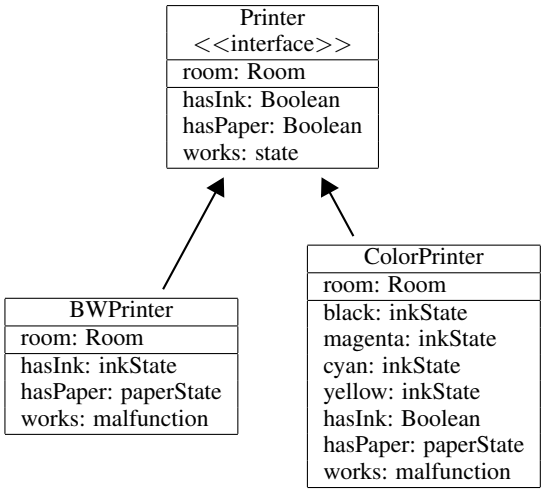


Figure 5: Two implementations of an interface.

Fig. 5 shows an example of an interface implementation, where the two classes *BWPrinter* and *ColorPrinter* implement interface *Printer* (which is no longer a class for this example). The *Printer* interface defines the minimal set of attributes and references any printer must declare: a reference to its room, whether it has ink, paper and whether it is working.

Fig. 5 is an alternative representation of classes using a UML syntax. Such syntax is necessary to point out attribute's and reference's types. It is more concise than the traditional representation of PRMs, i.e., the class dependency graph, see (Getoor et al., 2007). As already said, when creating a class, there is no need to know the dependency structure of the other classes to which it is related: only attribute and reference types are necessary for this task.

3.3 Multiple inheritance

Multiple inheritance is one of the major issues when defining an object-oriented formalism. The problem arises when diamond-shaped inheritance appears, as illustrated in Fig. 6. An ambiguity results from how the properties of class A are inherited by class D since two distinct paths exist from A to D (through B or C). Furthermore, if some properties of A are overloaded in B and C, which one should be inherited by D? Such issue can be dealt by using inter-

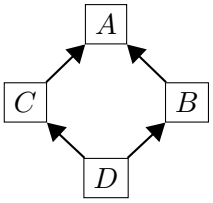


Figure 6: A diamond-shaped inheritance graph.

<state>works	*OK*	*Broken*	*malfunction*
OK	1	0	0
NOK	0	1	1

Table 1: The CPT of *BWPrinter.works* cast descendant which is of type *state*.

faces: since an interface only declares the existence of properties, each class implementing a given interface must declare and define itself those properties. The major drawback of this approach is that there is no reuse of properties definitions, i.e. there is code duplication. Another solution consists of explicitly declaring from which superclass a given property is inherited. But this proves to be cumbersome and bug prone. For this reason, we chose to use the interface-based solution. In addition, the notion of an interface is well suited for the PRM framework. Note that a class can implement as many interfaces as it designer wants to.

4 Attribute and reference overloading

In object-oriented programming languages, overloading is used to modify inherited properties. This is exactly what PRM attribute overloading and reference overloading do. In section 3.1, we showed how attribute overloading could be performed using inheritance. Now, by adding attribute typing, it becomes possible to also overload attribute's types.

4.1 Type overloading

People familiar with PRMs will remark that, in Fig. 5, if a class has a dependency over *Printer.works* it expects an attribute of type *state* and defines its conditional probability tables accordingly. However, connecting such a class to an instance of *BWPrinter* results in an incoherent probability distribution since the attribute referenced is of type *malfunction*. To fix this kind of problem we need the concept of *cast descendants*.

Cast Descendants are automatically generated attributes which are used to cast beliefs of an attribute into one of its super type. By exploiting Domain Generalization Functions (DGFs), it is possible to use deterministic tables to obtain beliefs with the correct domain size. Tab. 1 shows the conditional probability table of *BWPrinter.works* cast descendant, which casts type *malfunction* into type *state*. Algorithm 1 illustrates more formally how cast descendants are generated. The *goal* variable is the overloaded type and *a* the attribute whose type is a subtype of *goal*. The algorithm simply adds children to *a* until the goal type is reached. Procedure *generateCastCPT()* uses DGFs to generate deterministic tables as shown in Table 1.

```
Data: Type goal, Attribute a
Type t = a.type;
Attribute current = a;
while t ≠ goalType do
    Attribute child = new Attribute();
    child.type = t.super;
    child.cpt = generateCastCPT(t.super, t);
    current.addChild(child);
    current = child;
    t = t.super;
end
```

Algorithm 1: Cast descendant generation.

4.2 Reference overloading and instantiation

As seen previously, it is possible to define references in classes as well as in interfaces. We have shown how interfaces can be used to define probabilistic dependencies and since we introduced class inheritance and interface implementation, we can obviously use reference overloading. Given two classes X and Y, if Y is a subclass of X and if there exists a reference ρ in X referencing a class Z (or an interface I), then Y can overload ρ with a reference referencing any subclass of Z (or referencing any implementation of I).

Instantiating a reference amounts to linking it to an instance of the correct class in a given system. Given an instance x of class X and a reference $x.\rho$ referencing a class Z (or an interface I) $x.\rho$ can be instantiated in any instance of a subclass of Z (or any instance of a class implementing I). Class inheritance, interface implementation and cast descendants guarantee the existence of attributes de-

fined in a class (or interface) in any of its subclass (or implementation), which is sufficient to ensure a coherent probabilistic distribution.

5 Parfactor representation of PRMs

A large part of the statistical relational learning community has chosen first-order probabilistic models as their main framework. Actually, the only exact inference algorithm for first-order probabilistic models is lifted probabilistic inference (de Salvo Braz et al., 2005) and (Brian et al., 2008). Parfactors are the common formalization used in these approaches. It is important to note that, like most first-order probabilistic models, parfactors are more generic than PRMs: they can be used to represent complex systems impossible to represent using PRMs. However, they are less suited for modeling large-scale systems. Hence it is useful to be able to express PRMs in such a formalism. We will give a short definition of parfactors, or parametric factors, as they are given in (Poole, 2003).

Definition 7 (Parfactor)**.** A parfactor is a triple $\langle C, V, t\rangle$ where C is a set of constraints on parameters, V is a set of parametrized random variables and t is a table representing a factor from random variables of V to $\Re^+$.

Algorithm 2 details formally how an attribute can be converted into a set of parfactors. We will detail this algorithm using attribute *ColorPrinter.works* of Fig. 4(b). Since we represent PRMs, a parfactor's table will always be the conditional probability table of an attribute. Classes and instances are represented as parameters of parametric random variables, which are the equivalent of attributes in the PRM formalism. To ensure the exact representation of the structure encoded by the classes of a PRM, it is necessary to use two different types of constraints.

```
Input: Class c, Attribute attr
Output: Parfactor fctr
Parfactor fctr;
Add a isA() constraint over c's type;
Add a parametrized variable named by attr and prefixed
by attr's type;
foreach parent prt of attr do
    if prt not in A(c) then
        Add a isA() constraint over prt's class type;
        foreach reference ρ in the slot chain from attr to
        prt do
            Add a relational constraints in fctr
            matching ρ;
            Add a isA() constraint over ρ range type;
        end
    end
    Add a parametrized variable named by prt and
    prefixed by prt's type;
end
Copy in fctr's table attr's CPT;
return fctr
```

Algorithm 2: Parfactor generation of an attribute.

To represent classes, class inheritance and interface implementation we will use *isA()*-like constraints (e.g. *isAPrinter(X)*). Relations can be expressed as binary constraints in which each parameter has a *isA()* constraint (e.g. $room(X, R) \wedge isAPrinter(X) \wedge isARoom(R)$). The *ColorPrinter.works* attribute in Fig. 4 can be represented by the following parfactor:

$$\langle \{isAColorPrinter(X) \wedge isARoom(Y) \wedge isAPowerSupply(Z) \wedge room(X,Y) \wedge power(Y,Z)\}, \{malfunction\_works(X),\ paperType\_hasPaper(X),\ Boolean\_hasInk(X),\ state\_works(Z)\},\ t\ \rangle$$

The first part of this parfactor is composed of type constraints (*isAColorPrinter()*, *isARoom()* and *isAPowerSupply()*) and relational constraints (*room()* and *power()*). The second part contains the dependencies of the parfactor, which are the parametrized random variables *malfunction_works(X)*, *paperType_hasPaper(X)*, *Boolean_hasInk(X)* and *state_works(Z)*. The Cartesian product of their values is mapped to the values in *t*, which represent the CPT of *ColorPrinter.works*.

The *isA()* constraints encode the inheritance scheme of a PRM if, for each instance i of a system, a grounded variable is declared for each type of i, i.e. for all of its super classes and implemented interfaces. For example, an instance *coloria* of the *ColorPrinter* class will be represented with the following grounded variables: *isAColorPrinter(coloria)* and *isAPrinter(coloria)*.

Finally, cast descendants can be represented by including types names in the parametric variables declarations. For example *malfunction_works(X)* stands for the attribute *ColorPrinter.works* of type *malfunction*. Then, by generating parfactors for each cast descendant, the constraints names will ensure the correct structure. For example, the cast descendant *ColorPrinter.works* will be declared as:

$\langle \{isAColorPrinter(X)\}$
$\{state\_works(X),\ malfunction\_works(X)\},\ t \rangle$

At first sight, such a representation seems cumbersome but it illustrates the expressive power of parfactors and of first-order probabilistic models. First-order logic can be used to express very complex relations: only two types of constraints are necessary to represent all the notions presented in this paper. However such expressive power has a major flaw as semantics and relations are hidden in the mass of constraints declarations. When dealing with large-scale systems, creating and maintaining such knowledge base can be extremely difficult. PRMs with the strengthened object-oriented aspect we proposed here are a proposition to manage such knowledge with a formalism less expressive but much more scalable.

6 Conclusion

We proposed a strong object-oriented representation of PRMs by introducing interfaces, attribute typing, type inheritance, attribute and reference overloading. Such notions strengthen the expressive power of PRMs when dealing with structured and known systems. In addition, we have shown how PRMs with these features can easily be represented as parfactors, closing a gap between PRMs and more recent first-order probabilistic models. Strengthening the object-oriented features of PRMs enables a better representation of complex systems as well as the creation of new models in fields such as troubleshooting, reliability and risk management, where such models were often difficult to represent until now. Parfactors are used in the state-of-the-art lifted probabilistic inferences. Enabling the expression of PRM models into such formalisms will help comparing different first-order probabilistic implementations. However there is still room for improvements, especially for the graphical representation of PRMs and the implementation of user-friendly tools for model design and maintenance. Finally, the perspective of exploiting hierarchical knowledge can lead to new inference algorithms in PRMs.

Acknowledgments: this work has been supported by the DGA and has benefited comments, suggestions and ideas from the SKOOB consortium members (http://skoob.lip6.fr).

References

O. Bangsø and P.-H. Wuillemin. 2000a. Top-down construction and repetitive structures representation in Bayesian networks. In *Proc. of FLAIRS 2000*, pages 282–286.

Olav Bangsø and Pierre-Henri Wuillemin. 2000b. Object Oriented Bayesian Networks: A framework for topdown specification of large Bayesian networks and repetitive structures. Technical report, Department of Computer Science, Aalborg University., Aalborg, Denmark.

Milch Brian, Luke S. Zettlemoyer, Kristian Kersting, Michael Haimes, and Leslie Pack Kaelbling. 2008. Lifted probabilistic inference with counting formulas. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 1062–1068.

R. de Salvo Braz, E. Amir, and D Roth. 2005. Lifted first- order probabilistic inference. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 1319–1325.

L. Getoor and B. Taskar. 2007. *Introduction to Statistical Relational Learning*. The MIT Press.

Lise Getoor, Nir Friedman, Daphne Koller, Avi Pfeffer, and Benjamin Taskar. 2007. Probabilistic relational models. In L. Getoor and B. Taskar, editors, *An Introduction to Statistical Relational Learning*. MIT Press.

Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models*. The MIT Press.

D. Koller and A. Pfeffer. 1997. Object-oriented Bayesian networks. In *Proceedings of the 13th Annual Conference on Uncertainty in AI*, pages 302–313.

K.B. Laskey. 2008. MEBN: A language for first-order Bayesian knowledge bases. *Artificial Intelligence*, 172:140–178.

J. Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufman.

A.J. Pfeffer. 2000. *Probabilistic Reasoning for Complex Systems*. Ph.D. thesis, Stanford University.

David Poole. 2003. First-order probabilistic inference. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 985–991.

Xiang Yang. 2002. *Probabilistic Reasoning in Multi-Agent Systems: A Graphical Models Approach*. Cambridge University Press.